Visualization of CVS Repository Information

Xinrong Xie, Denys Poshyvanyk, Andrian Marcus*
Department of Computer Science
Wayne State University
Detroit Michigan 48202
{xxr, denys, amarcus}@wayne.edu

Abstract

Mining software repositories is an important activity during software evolution, as the extracted data is used to support a variety of software maintenance tasks. The key information extracted from these repositories gives a picture of the changes on the software system. To have a complete picture, tailored to the needs of the developer, the extracted data needs to be filtered, aggregated, and presented to the users.

In this paper we propose a new visualization for such data, which relies on an existing software visualization front-end, SourceViewer3D (sv3D). The new visualization allows users to define multiple views of the change history data, each view helps answer a set of questions relevant to specific maintenance tasks. Data can be viewed at different granularity (e.g., file, line of text, method, class) and comprehensive views can be defined, which display to the user multiple data types at the same time. Complex questions and tasks can be solved with the aid of such views.

Note: the paper uses many colors. We recommend reading an electronic (color) version.

1. Introduction

The use of version control systems in software management is a standard practice nowadays in almost any software development project. Not only that they support users to collaborate effectively, the data stored by these systems on the history of changes has many uses in the maintenance of software systems.

For instance, a project manager may use information from the change history to assign new tasks to the most appropriate developers; a tester may find out who is responsible for a bug and when it was injected in the source code; a developer may find out what parts of the system were changed during the implementation of certain features; etc. The software repositories keep the data necessary to formulate answers to all these and other questions, which directly support a variety of maintenance tasks, such as impact analysis [2], improving software design [19], refactoring [5], guiding software changes and verifying the completeness of a change [27], detecting logical coupling [10], etc.

Researchers and practitioners devised a variety of methods to extract and analyze information from software repositories [13]. The larger the evolving software systems and their development teams are, the larger the repositories grow, and the more important the collected data is. As with any analysis problem of large data sets, data visualization is often used to show the users information obtained through mining the software repositories [1, 3, 4, 6-9, 12, 15, 22, 23, 25, 27]. Individual tools provide visualizations, which help the user to find the answers to a subset of their questions.

This paper presents a new tool named **C**VS**V**iewer3D (cv3D), which extracts, processes, and visualizes information from CVS repositories. The visualization component is based on an existing framework, SourceViewer3D (sv3D) [18], which can generate 2D and 3D views composed on easy to understand, abstract geometric metaphors. Using sv3D to visualize CVS information was suggested before by researchers [10].

Several features set cv3D apart from related work. The tool allows the user to define views of the data at different granularity level (e.g., system, file, method, line); navigation between granularity levels (and views) is supported. Within each view, various version centric perspectives can also be used. Finally, taking advantage of the 3D rendering space and metaphors, different visualizations can be combined into comprehensive views, which help the user to answer an extended set of maintenance related questions.

The new tool is analyzed and compared with the existing state of the art in the field; usage scenarios and examples are presented.

2. Related work

A recent survey on the use of visualization to support awareness of human activities in software development [20] reviews many of the existing tools that visualize information extracted from software repositories.

* Contact author

Existing tools use different types of visual metaphors to represent heterogeneous data. For those tools that will be used later in the paper for comparison, we provide a three letter abbreviation in parenthesis, for easier reference later on.

Seesoft (SEE) [7] is one of the earliest visualization tools for version control system, which represents source code through file and pixel maps. Color is used to show the data extracted from the repository. Other tools were developed following the Seesoft approach. Augur (AUG) [8], to name one, visualizes code artifacts by lines and encodes two additional data elements next to the line information. It also provides secondary views to show various cumulative graphs and statistics. CVSscan (CSS) [22] is another example that uses file maps, where each version is represented by a column and the horizontal axis marks the time. It also has a separate view, which shows various metrics along with the source code of a system and implements a version centric filter.

Charts and matrixes are also used to display the evolution of the software. Evolution chart (EVC) [10] shows a property on the vertical axis and time on the horizontal axis. This approach is suitable to explore evolution of one property using one software entity. In order to view multiple properties and compare evolution of different entities, classes are arranged using the Evolution Matrix (EVM) [15]. In the evolution matrix, each rectangle represents a version of a class and each line holds all the versions of that class.

Graphs are also used to reveal the hierarchical evolution of software. For example, Xia/Creole (XIC) [25] uses hierarchical graph views to display architectural differences between artifacts of two versions. In the graph-based views, visual attributes, such as color and position are mapped to the data extracted from CVS. Beagle [21] displays every release of a project in call graph. It also generates additional static graphs: tree views to show how a given method evolves and scatter plots to show the structural changes of a file or group of files. Another visualization tool, softChange (CHA) [12], is composed of a graphical component and a hypertext component. The graphical component provides two types of views: histograms with statistical information and graphs representing files, authors, and their inter-relationships. The hypertext component in softChange allows the user to navigate, search and inspect source for a given change.

Other 2D representations have also been used to visualize the change of software and related information: spectrographs [24], fractal figures (FRF) [4], etc.

Since the historical data extracted from version control system usually consists of large amount of information with multiple attributes, some tools use three-dimensional visualizations. The third dimension is usually used to express the time dimension. In VRCS [14], each version of a system within a history is represented as a 3D tree showing module and file relationships along the x and y axes, and time on the z axis. Each major release is represented as a sphere and links between version nodes show which ones should be compiled together. Gall [11] proposed a visual representation for examining a software system's release history using color and third dimension. Colors are used to display module properties and their historical changes in the system. The historical information is represented on the third dimension, which stands for time.

Several tools are trying to combine visualization techniques. For instance, Advisor (ADV) [6] constructs multiple views, using matrix displays, 2D and 3D bar charts, pie charts, zoom-able text displays, and graphs to express the same software change data. Another tool, JRefleX (JRX) [23], shows analysis results in matrix, pie, and 2D bar charts, etc.

Some tools are trying to solve special change history problems with visualization. ROSE (ROS) [27], to name one, generates a set of suggestions which are based on history of related changes and sorted in a table by support count and confidence values, obtained for every related software entity which is likely to change. CCVISU (CCV) [1] implements co-change visualization with a clustering layout.

3. Extracting and representing historical data

The main architectural elements of cv3D are presented in Figure 1. First, cv3D extracts the data from a CVS repository. After that, the extracted data is processed for further analysis. Additional data, statically extracted from the source code is also added. Once the data is extracted and preprocessed, the user can request sv3D to represent the desired information using a set of predefined views, discussed in the next section. The user's query is translated into a set of commands to extract the necessary data and display it with sv3D. Finally, once sv3D renders the data, the user can interactively explore all the details of the visualized historical information. The user can also define new views by redefining the default mappings between the data and the visual attributes of the metaphors in sv3D.

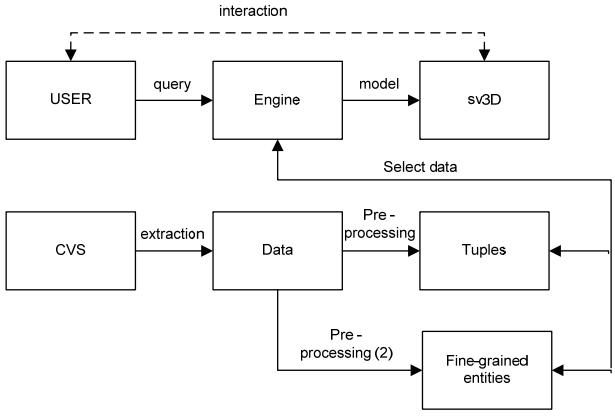

Figure 1. cv3D architectural elements

Once the data is extracted from the CVS repositories, the pre-processing is a one time operation. The pre-processed data is stored in an XML file and subsequently loaded into sv3D.

The data extraction and pre-processing part is decoupled from the visualization in cv3D, so that we are able to adapt our parsers and other parsers to the sv3D framework. In the current version of cv3D, we extract information from CVS repositories. Extracting data from other version control systems, like Subversion, requires implementing additional parsers and adapters for our data model, which is not a complicated issue.

3.1. Extracting the CVS data

In order to visualize historical data for further analysis, we extract the data from CVS repository, formally defined as R, which stores all versions of the files for a given software system under version control. A repository R is a set of n unique files: $R = \{F_1, F_2, \ldots F_n\}$.

In turn, every file F_i has a number of revisions in the repository, $rev(F_i) = \{F_{i1}, F_{i2}, \ldots F_{ir}\}$, where $r = |rev(F_i)|$, the number of revisions for file F_i.

In order to extract information about the changes according to every two consecutive revisions of the same file, cv3D uses the UNIX *diff* tool, which computes the difference between $F_{i,j}$ and $F_{i,j+1}$. Then, the result is parsed to determine the change status (*added*, *deleted*, *modified, or constant*) for every line in revision F_{i+1}.

For every revision of the file F_{ij} we obtain the set of the following tuples:

F_{ij} = <file, j, author, date, LOC_status>, where *file* is name of the file F_i, *j* is the current revision of the file F_i, *author* is the name of the author who contributed to (committed) this revision to the repository, *date* is time and date when this revision was committed, and LOC is the change status for every line in this revision.

In order to extract the CVS comments and associate those with respective revisions of the file we used *cvs log* command, which produces the required mapping between CVS comments and code:

F_{ij} = <rev_id, cvs_comments> for all i = 1 ... n and j = 1 ... $|rev(F_i)|$, where n is the number of unique files in the repository and $|rev(F_i)|$ is the number of unique revisions of every file.

By associating CVS comments with file revisions, we get the final tuples:

F_{ij} = <file, j, author, date, LOC_status, CVS_com>, where *CVS_com* is the CVS comment associated with the current revision of this file.

3.2. Data pre-processing for fine-grained entities and co-changes

By querying the CVS repository it is possible to obtain information only on files and differences between revisions of files, as CVS does not provide information on specific syntactic entities (methods, functions, classes, etc.) that have been changed. In order to be able to analyze fine-grained entities we need a different type of preprocessing (see Figure 1).

Changes made within fine-grained entities like methods or classes can be determined using a *diff*-tool and a light-weight analysis used in this paper (determining the lines of text where methods start and end). Using this approach we can determine actual changes and types of changes (changed, added, deleted, or constant) in methods and classes in two consecutive revisions of files. However, this approach depends upon the quality of the actual *diff*-tool with intrinsic problem of the use of *diff*-tool which does not take renaming or moves between files into account. Another approach, which first computes actual locations of entities in the source code and then compares those entities in two revisions, is considered to be more precise, but more complicated to implement [26].

Since CVS does not keep track of which files have been changed together within the same commit operation (or group commit operation), we group commit operations in time in order to analyze files or syntactic entities that have been changed together. There are two available approaches in the literature on how to group related commit operations: fixed time windows and sliding time windows [26, 27]. For the purpose of flexibility, we use sliding time windows approach to group related transactions.

In its current form, cv3D extracts and computes the following data and measures:

- Developer (author), Change status, Version, Time of change, Comments;
- *#LOC, #methods per class, #revisions, #LOC changed, Co-change (support, confidence), and #changes per author*.

Some of this data is directly extracted from the repository, while the measures (in italics) are derived. This data is then mapped to the sv3D [18] visual metaphors and their attributes. sv3D provides users with rich interactions, such as: rotation, scale, and zoom of the entire space and of individual containers; moving containers; use of transparency for filtering, etc.

3.3. The sv3D data model

sv3D is a software visualization front end, which extends the pixel map metaphor in three dimensions. Its application *P* is defined as a quadruple $P = \{V, D, S, M\}$, where *V* defines the visual metaphors to be used, *D* represents the data resulted from software analysis stored as a set of files $D = \{d_1, d_2, \ldots, d_n\}$, corresponding to a set of source code files $S = \{s_1, s_2, \ldots, s_n\}$. $M = \{m_1, m_2, \ldots, m_k\}$ defines the mapping between data and visualization as a set m_i of relations $m_i = D*S*V$.

In the current version of sv3D each attribute is linked to an element of the visualization $v \in V$, by a mapping $m_i \in M$. Currently sv3D supports mapping to the following elements of the visualization, defined in V:

- Poly cylinder – p; Container of poly cylinders – o;
- Poly cylinder position in the container on its o_x axis – p_x;
- Poly cylinder position in the container on its o_y axis – p_y;
- Poly cylinder height – z_+; Poly cylinder depth – z_-;
- Poly cylinder color on o_{z+} axis – c_+;
- Poly cylinder color on o_{z-} axis – c_-;
- Poly cylinder shape – σ

Every element in V is a nine-tuple:

$v = \{p, o, p_x, p_y, z_+, z_-, c_+, c_-, \sigma\}$

Views in sv3D are defined by mapping data elements from any d to the visual elements and their attributes from v. A view may have several containers and all elements from d need not be represented in a view, nor need all elements of v to be used.

4. Answering questions and defining views

As mentioned before, visualization of CVS data is important as it helps answering important questions about the evolution of the software. A representative set of questions addressed by most existing tools is summarized in bellow:

1. Who has been working on the artifacts?
2. Which authors work on the same files?
3. What kind of modifications did the author make?
4. When was a modification made?
5. Why was a modification made?
6. What files do frequently change together (co-change)?
7. Which parts of the code are stable?
8. Which parts of the code change frequently?
9. How many authors worked on a release of a system?
10. How many files or lines are added or deleted?

Since software engineering is a highly collaborative activity, one of the major issues is to "understand the activities of others, which provide a context for one's own activity" [20]. Usually, developers need to know who else is working on the project and what exactly they are working on, which files they have been modifying etc. Questions 1 and 2 are geared towards supporting understanding the "programmer's neighborhood" in a software project.

Questions 3-5 relate to understanding fundamental components of a software change: when, why and what exactly has been changed? Every of these questions require different information about the change. For example, in order to understand why the change has been made we may need to analyze the semantics of the change between consecutive revisions as well as CVS comments associated with particular change. Answering the "when" question is rather trivial since every revision of the file has a time stamp, which is easily extracted. On the other hand, the "what" question is two-fold, since we may identify the actual change on different granularity levels: simple lines of code changed using *diff* or fine-grained entities using syntactic (based on an

Table 1. Summary of tools visualizing CVS data across different views, metaphors, and questions they help to answer. The tool abbreviations are those defined in section 2.

Views	Questions	Visual Metaphors	Tool	Mappings and data represented
System	10	Bar, Data Sheet	ADV	# of lines changed
	10	Chart	EVC	# of programs added/deleted
	10	Matrix	JRX	Area of bubble – number of changes
	9, 10	Bar	CV3	See section 4.1.1
File/ Class	1,2,7,8	Fractal	FRF	Size - # submissions per author
	1,3,4,5, 7,8,10	Graph, Text Nested Graph	XIC	Color – author name; LOC in text; Position of node – ordered by last commit date; CVS comments; Size of node - # of changes
	3,7,8	Bar	JRX	Percentage of bar - # of type of modification
	4	Matrix	JRX	Show the type of change and time in Matrix
	6	Table	ROS	Show the sorted co-change files in table
	6	Graph	CCV	Color – subsystem name, File – filled circles
	7,8	Matrix	EVM	# attributes – width of node; # methods – height of node
	4,7,8	DTF	DTF	color – # of revisions/bugs for period of time
	10	Chart	CHA	Percentage of bar - # of type of modification
	1,2,4-8,10	Bar	CV3	See section 4.1.2
Function/ Method	3,7,8	Bar	JRX	Percentage of bar - # of type of modification
	7,8	Bar	CV3	See section 4.1.3
Line	3,4,5,7,8	Text, Tooltip, Nested Graph	XIC	Show lines of source code in text; Color – time; Shows CVS comments and documentation; Size of node - # of changes
	3,7,8	PixelMap	SEE AUG	Color – author last submitted; Color – age of code
	3	PixelMap	CSS	Color – change status
	3,4,5,7,8	Bar	CV3	See section 4.1.4

abstract syntax representation of the source code) or semantic differencing [16].

One of the important applications of software change data is using that to detect evolutionary or logical couplings [10, 27]. One of the applications of evolutionary coupling is to use historical change data to predict likely changes or prevent errors or incomplete changes (question 6).

Another important question for software evolution activities is to know which software components are stable (question 7) vs. those that repeatedly need corrective maintenance because of decay, perfective, or adaptive maintenance (question 8).

In addition to these issues, users need views to see software release level details, e.g., the size of changes in any given release (question 10) [6]. It is often useful to identify developers who contributed to particular release of the software (question 9).

The extracted data needs to be grouped into views in order to answer specific set of questions. For example, question 9 and 10 ask about the changes in the entire software system, whereas question 2 refers to specific files. Questions 7 and 8 can be answered by visualizing the change frequency information for methods in classes, while question 3 reveals modifications to every line of code in the file. Furthermore, some questions may be addressed using more than one granularity level, e.g. questions 7 and 8.

4.1. Views

In order to properly address all these questions we define a set of *granularity-based views* for the visualization of CVS data: system level view, file (class) level view, function (method) level view, and line (LOC) level view. The lowest level granularity software element that is visualized defines the abstraction level of each view. Each view is used for a specific set of questions, to answer some questions, several views may be combined.

While defining such views at various granularity levels is not uncommon in software visualization problems, the CVS data has an important aspect, not common in other software visualization tasks: *time dependency*. This fact requires us to define *version-centric perspectives* for each of the defined views. For example, users may need to investigate only one revision at a time, a sequence of revisions (of the same software elements), or the differences between two versions.

These views are general and they are realized differently by various existing tools. Most tools support the representation of the data in one or two views. In cv3D we are providing the users the possibility of defining all four types of views and perspectives.

We present how these views are constructed in cv3D and other tools and which questions are addressed by these views. We define some of the possible mappings for each view however; the user is able to select his preferred mappings as well. Table 1 summarizes this information.

4.1.1. System level views

System level views represent different releases or versions of the entire system to overview the evolution of the software. Although we discuss here only two questions related to the system views, it is not limited only to these two.

In cv3D the system views can be realized in several ways. The simplest mapping to use is by representing a system release by a poly cylinder. Since we only need to capture the order of the release, they need not be grouped in any specific fashion (other than chronological), hence this can be represented as a set of 2D bars (see Figure 2). Using system view we may represent the following data - number of files/classes/methods added or deleted, number of revisions, number of authors, LOC of changes. Color and height can be mapped to variety of data elements or measures.

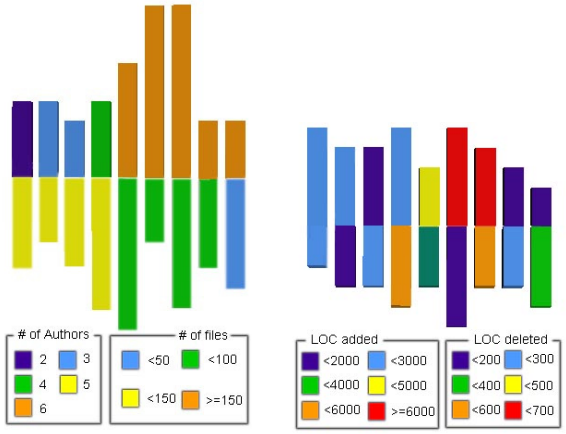

Figure 2. System level view. The entire system is represented, with each bar (flattened poly cylinder) showing one version. Two alternative mappings of the visual attributes to data are shown.

Figure 2 shows two possibilities of such mappings. On the left hand side, nine consecutive versions of the K-Meleon system are represented. Each bar (flattened poly cylinder) shows one such release. The height of the poly cylinders (bars in this case) shows the number of lines added in each version and the color shows how many authors contributed to that release. The depth (negative height) of each bar shows the number of lines deleted in each version (times ten here for scaling purposes). The color here shows the number of files in that release.

On the right hand side, we show an alternative mapping that the users can define. The same nine versions are represented. This time, the height and the depth show the (same information as above, but) number of files that are modified with added and deleted lines of code respectively, and colors show on each dimension the number of files in each version that were modified, added or deleted respectively.

System views support all three types of version-centric perspectives. The example in Figure 2 shows multiple versions of the system at the same time.

System views are used to address questions 9 and 10. Question 9 is about the information of every single version, thus the number of authors could be mapped to height. On the other hand, Question 10 is about the difference between two versions, thus the number of files/classes/methods/LOC added or deleted is mapped to height. For these answers, different perspectives can be used (i.e., only one version or two versions to be compared).

Three other tools (see Table 1) can answer question 10 using system view. ADVIZOR uses bar chart and data sheet to show how many lines of code is changed in each version. Evolution chart display the chart of how many programs is added or deleted of the whole system during the evolution. And JRefleX uses area of bubbles to represent the number of changes in each version.

4.1.2. File (class) level views

The bulk of change history information can be represented using file level views. We also call this view *class view* when we display change history data for classes, rather than files, as cv3D extracts this information from the source code (not directly available in the CVS repository).

File views in cv3D use the following sv3D mappings: container represents a system and a poly cylinder represents a file/class in the system (see Figure 3). Using file views we are able to represent the following data – number of LOC changed, authors of the changes, time, comments, and co-change (support and confidence). Once again, different version-centric perspectives can be defined. Figure 3 aggregates data from all the revisions of the system we studied.

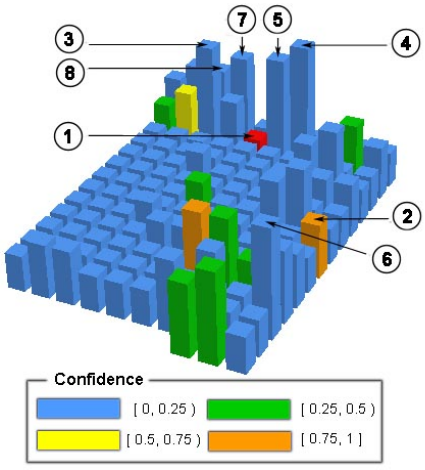

Figure 3. File level view representing the files that co-changed with BrowserViewFind.cpp (the red poly cylinder) in the K-Meleon system. Each poly cylinder shows a file. The support count values are mapped to height and confidence values to color.

File (class) views are used to answer questions 1, 2, 4-8. Question 1 is about showing information of all single revisions. Author name is mapped to color. Questions 4, 5 are about every revisions of the file. Time and comments will be shown in text. Question 6 is about summary results of all revisions. Confidence value is mapped to color and Support Count is mapped to height (see Figure 3). Questions 7 and 8 are about summary of all revisions. LOC/method changed is mapped to height.

Many tools are able to answer a subset of these file level granularity questions using different visual metaphors and mappings (see Table 1). However, none of them can answer all the questions. For example, Xia/Creole can answer questions 1, 3-5, 7,8 and 10. It answers question 4 by ordering the position of nodes, each of which stands for a revision of a file. It may also answer question 10 using the size of the nodes representing the number of changes. Another visualization tool, Fractal Figure, is able to answer question 1, 2, 7, 8, using fractal figures, whose size representing the number of submissions per author. Evolution Matrix, which uses nodes for classes, number of attributes for width of nodes, and number of methods for height of nodes, is able to answer question 7 and 8. CCVISU, a visualization tool for co-change, have filled circles for files, and color for subsystem name, displaying a force-directed graph layout, may help to answer question 6.

4.1.3. Function (method) views

In function (method) level views, the representation in cv3D is as the following: container represents a file (or class) and a poly cylinder represents a function (or method) in the file; data that can be mapped to the attributes - # lines changed, number of methods added or deleted, etc (see Figure 4b). Once again, different version-centric views can be defined. Figure 4b shows the difference between two versions.

Function views are used to address questions 7 and 8. Questions 7 and 8 are about summary of all revisions. LOC/method changed is mapped to height.

Few tools are able to deal with questions at method level granularity (see Table 1). This is no surprise as most tools do not add data extracted from the source code to the one extracted from the CVS repository. JRefleX can also answer questions 7 and 8 using bar charts.

4.1.4. Line (LOC) level views

In the cv3D line views, the representation is as the following: a container represents a file, a poly cylinder represents a line of text in the file; author name, change status, revision number, date and comments can be mapped to the visual attributes (see Figure 4a). In Figure 4a, we can see several versions of the same file from the system.

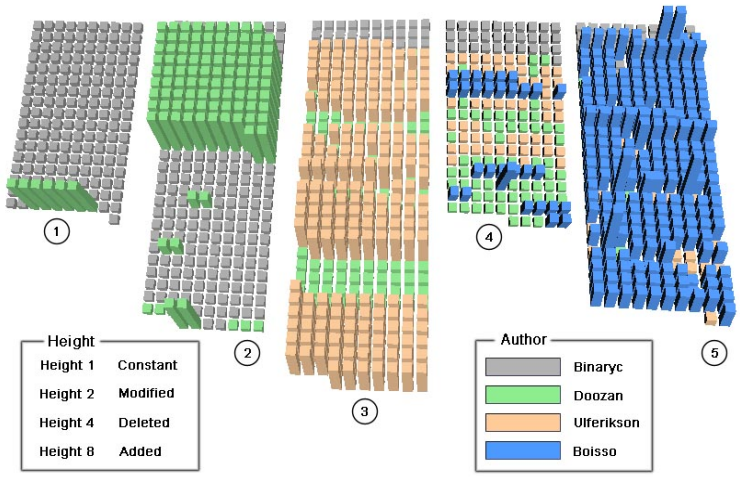

Figure 4a. Line view with 5 revisions of BrowserViewFind.cpp. The author is mapped to color and change status is mapped to height for each LOC/poly cylinder.

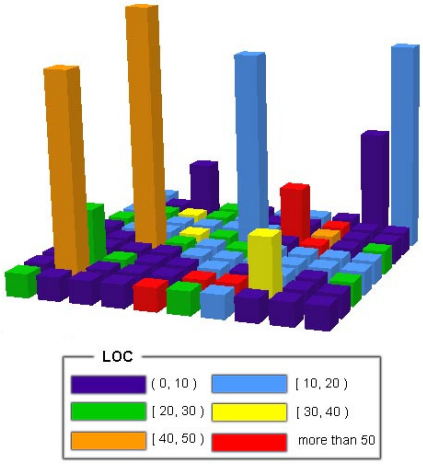

Figure 4b. Method view representing the BrowserView class with its methods. The #LOC in each method is mapped to color and the number of changed lines (from version 1.114 to 1.115.) to height.

Line views are used for questions 3, 4, 5, 7, 8. Question 3 is about difference between two revisions. Change status can be mapped to height, and author is mapped to color. Questions 4 and 5 are about every revision of the file. Time and comments will be shown in the text. Questions 7 and 8 are about difference between two revisions. Change status is mapped to color.

Several tools display or visualize lines of code issue (see Table 1). For instance, Seesoft and Augur represent lines of code as pixels or lines in file maps, using color for author name for question 3 and color for the age of code for questions 7 and 8. CVSScan also visualizes the difference of lines between two revisions in file maps using color for change status. Xia/Creole displays text information for questions 3, 4, 5 and uses nested graph for questions 7 and 8.

5. Using cv3D in software evolution

In order to show how the cv3D generated views can be used to solve problems related to software maintenance, we used cv3D to solve several tasks. We give here two examples of usage. In the first example we use cv3D visualization of historical information to guide software changes, whereas in the second example we look into using visualization to support more general activities, which may be useful in a larger number of maintenance tasks.

In both studies we visualize historical information of the same open source system, namely K-Meleon (http://kmeleon.sourceforge.net/), which is a Win32 web browser, powered by the same Gecko rendering engine used by the Firebox and Mozilla browsers. The software consists of about 185 *.h and *.cpp files with 57 classes and 213 methods. The latest release of K-Meleon 0.9.12 has 34,253 lines of code and 6,940 lines of comments (not included in the LOC). Overall, the history of the project is quite rich, consisting of more than 1,000 file revisions (across all the files in the project). For this study, we extracted and analyzed the version history of the K-Meleon project up to the release 0.9.12. Version numbers of files in the CVS repository are not related to the Sourceforge release number.

5.1. Visualizing historical information to guide software changes

Zimmermann et al. [27] proposed an approach to guide software changes by mining version histories. This approach allows suggesting and predicting likely changes as well as preventing errors due to incomplete changes. For a software entity that the programmer is currently investigating, the tool (i.e., ROSE), generates a set of suggestions which are based on the history of related changes and sorted by support count and confidence values obtained for every related software entity which is likely to change [27]. These suggested entities for further exploration are presented in simple tables.

In this part of the study we show how sv3D can be used to enrich the process of guiding software changes using aforementioned approach with visualization. In this case, we use visualization to represent the information about co-changed software entities using file level granularity. The advantage of using visualization in this case is that we can represent the related files, which were co-changed in the past, in the context of the complete software system (in other words, as the map of previous co-changes).

In this example we focus on a specific maintenance problem: fixing a bug in K-Meleon. The bug that we chose relates to the "Find" feature of the browser which has a problem with auto-search. The problem with the

current implementation is that "Find" starts to search at the end of the current selection of the cursor, however, with auto-search (using "Find Next") it should start at the beginning of the page. We decided to choose this bug for the study since it has been fixed, thus we could use this information to verify the correctness of our fix.

We identified the place in the source code which implements the "Find" concept. The concept was found in the BrowserViewFind.cpp file, which contains the implementation of the *CBrowserView* class, which is responsible for handing the "Find" bar in the browser. We also located the method *OnFindNext*, which possibly needs to be modified in order to fix the bug. However, after we started to plan how to implement this bug fix, we also needed to perform impact analysis to see whether we need to change anything else to complete the fix. To support impact analysis we visualized historical information about co-changes of BrowserViewFind.cpp with other files in K-Meleon software system, using cv3D (see Figure 3).

In this case we used color and height to represent confidence and support count values for co-changed files. By using colors to represent confidence values it is easy to identify, for example, those files which were changed only together with BrowserViewFind.cpp. The color ranges that we use to represent similarities are outlined in Figure 3. We also represent support count by mapping frequencies of co-changes of the other files in K_Meleon with BrowserViewFind.cpp. In addition to the historical information represented with color and height attributes, the user can obtain more information about the method represented in cv3D by clicking on the respective poly-cylinder. In other words, the user may view the content of the given file as well as CVS comments listed for all revisions of the file represented as a poly-cylinder.

The file BrowserViewFind.cpp is represented with red color 1 in Figure 3. We started exploring the relevant files by looking to the poly-cylinders with the highest confidence values which are between [0.9, 1.0]. The file that catches attention at the very first sight is nsGenericFactory.cpp (2 in Figure 3), which has support count of 1 and confidence of 1. After clicking on the poly-cylinder, we noticed on the information tooltip that this file has just one revision, in other words it has been just created along with changing BrowserViewFind.cpp and does not actually relate to it.

The next obvious candidate is BrowserViewUtils.cpp (3 in Figure 3), which has a support count of 5 and confidence of 0.132. Again, after exploring the source code for this file we concluded that we will not have to modify this to fix the bug.

Then, we explored two poly-cylinders, BrowserView.cpp (4 in Figure 3) and BrowserView.h (5 in Figure 3) respectively. The BrowserView.cpp file has a support count of 6 and confidence of 0.053. It has been changed 115 times, which is a good indicator of its importance in the system. After exploring this file we decided that we need to change several macros in MESSAGE_MAP. Also, the file BrowserView.h needs to be changed since we needed to comment out several handlers for messages from the previous file. By browsing CVS messages for revision of the BrowserView files we found a message "small find related improvements" in revision 23. Inspecting the difference between revision 22 and 23 revealed that this was an unsuccessful attempt to fix the bug.

The Resource.h file (6 in Figure 3) has been changed 67 times overall and 4 times together with BrowserViewFind.cpp. We decided that this file will be changed as well, since we will need to add several definitions of constants for the visual elements in "Find" dialog box.

The next inspected file was BrowserFrameGlue.cpp (7 in Figure 3). It turned out that we needed to modify the login in the *FocusNextElement* method in order to correctly set focus for "Find" dialog.

Eventually, we explored BrowserFrm.cpp (8 in Figure 3) and we identified several methods that we need to change in order to complete the bug fix: *PreTranslateMessage* and *OnShowFindBar*.

Overall, we were able to successfully implement the bug fix in this case by exploring information about co-changed files represented in cv3D. We were able to quickly identify important evolutionary couplings between files and complete impact analysis for this change. Among other observations, we noticed that it would be useful to map additional information to color or height. For example, we could map to color the most recently co-changed files to BrowserViewFind.cpp. It would be helpful in the case when we know that we recently implemented a similar change and are going to explore the same classes or files this time as well.

This example shows that visualizing historical data about co-changes in the source code is useful in a number of ways. First of all, the user can easily grasp the overall image on how the files are related depending on the relative frequency of co-changes with a current file under investigation. Second, the user can easily navigate between different levels of abstraction by simply clicking on the poly-cylinder and looking into the source code of the necessary revision, differences between two different revisions, or even obtaining CVS comments for all the revisions of the chosen file. All this information is within the reach with few mouse clicks on the selected poly-cylinders, in the cv3D representation on the historical information of the software system. Using cv3D we can display all the instances of this particular co-change and the user can explore this information "back in time" by navigating those co-changes in different file revisions (as in Figure 4a) and reading specific CVS messages, which may provide clues on why this specific co-change happened. Obviously, we can visualize other aspects of

evolutionary changes that may be beneficial for the developer.

5.2. Studying evolution of a software system with cv3D

The main target group of cv3D is the maintenance community, since the maintainers perform different tasks, which are usually done on the source code which has been developed some time ago. Software maintainers usually spend a lot of time for program comprehension and finding out who is the expert on different parts of the system. We tailored our approach to support these tasks of maintainers.

In this example we decided to look closer into the evolution of the file BrowserViewFind.cpp from the K-Meleon project and see who did what kind of modifications (see Figure 4a). In order to effectively visualize the history of the file we map a container to the difference between two consecutive revisions of the file (e.g., container BrowserViewFind.cpp 1.2 shows revision 1.1 plus the difference which were introduced in 1.2; in turn 1.3 shows revision 1.2, which may have lines of code propagated from 1.1, plus the difference introduces in 1.3., and so on). In this view we map information about author to color and information about change status of the line of source code to the height. Since we have only 4 possible combinations for change status (unmodified, modified, deleted and added) we use the height of 1, 2, 4 and 8 respectively to represent all these cases (Figure 4a).

Container 1 (see Figure 4a) shows a very simple pattern: several poly-cylinders are elevated meaning that six lines of source code have been added to the very first revision of this file. By reading the tool-tips associated with these poly-cylinders we identify the "green" author of the modifications who is "doozan", whereas original version of the file belongs to the "grey" author, "bynaric". A similar pattern is observed for the second container 2. "Doozan" adds mode functionality to the file and modifies a few existing lines of source code. Using the cv3D control panel we explore the lines of source code which have been added to the revision 1.3 of the files. Additions are simply two new methods *OnShowFindDlg* and *OnFindMsg* and modifications involve "un-commenting" existing methods. In this case there are several lines of code (i.e., the green ones with height of 1) which were introduced by "Doozan" in revision 1.2, but remain unmodified in the current revision. The author is the same from revision to revision for these lines until a new author actually modifies these lines.

A new author, "ulferikson", is responsible for committing new modifications in revision 1.4 (container 3). It is clear that this revision contains only deletions and modifications of the existing source code.

Container 4 representing revision 1.5 reveals a pattern of fixing small changes. A new author "boisso", represented with the blue color, modifies several lines of source code and adds one more line. Quick inspection of the lines shows that all the changes relate to small memory allocation and data type conversion problems.

The revision that underwent the most serious modifications is 1.6 represented with container 5. It is obvious that almost everything has been modified in this revision, leaving almost no traces of the original authors of the files.

Overall, this usage example shows how this line level view can contribute to the understanding of some of the aspects of the evolution of chosen files. Using this view it is easy to determine who is the author of modifications in specific revision and what kind of modifications are performed in this revision. Using additional tools of cv3D, like the control panel, it is easy to explore actual source code and CVS comments of the chosen revision.

Another view that may reveal useful patterns for consecutive revisions is representing fine-grained software entities, e.g. classes and methods (see Figure 4b). Using this view we can quickly grasp the picture of how many methods and how many lines of code have been modified in those methods as well as how large those methods are, or other similar information. For example, in Figure 3 we can easily see that most changes (the tall poly cylinders) correspond to the smaller methods (color) with less than 50 LOC. Only one of the six largest methods (i.e., the red ones), *OnDragUrl*, was modified between these two revisions.

The methods which observed to be heavily modified are *NicknameLookup*, *OnFileOpen*, *GetUnicodeFromCString*, and *ChangeTextSize*. There are also some methods with smaller modifications: *OnUrlSelectedInUrlBar*, *OnNewUrlEnteredInUrlBar*, *OnDragUrl*, etc.

We can also conclude from this view that the *BrowserView* class has only a few methods which have more than 100 LOC, the majority of the methods have less than 20 LOC.

6. Conclusions and future work

We introduced a new tool, cv3D that visualizes information extracted from CVS repositories. Compared with other tools, cv3D can represent the data in views at different granularity levels, which may contain various version-centric perspectives. These views help the user answer many questions about the CVS data, which in turn directly support a variety of maintenance tasks.

The visualization component of cv3D is based on an existing tool, sv3D. As sv3D will evolve, some of its new features will also be used in cv3D.

Cv3D can be used to define more views, in addition to the ones mentioned and shown in this paper. We must investigate and assess these other views as well.

Usability studies [17] and further comparisons with other tools are planned and needed.

7. Acknowledgements

This research was supported in part by grants from the National Science Foundation (CCF-0438970) and the National Institute for Health (NHGRI 1R01HG003491). Louis Feng, Jonathan Maletic, Andrey Sergeyev, and Denise Comorski contributed to the development of the previous versions of sv3D.

8. References

[1] Beyer, D., "Co-Change Visualization", in *Proceedings 21st IEEE International Conference on Software Maintenance (ICSM'05)*, Budapest, Hungary, Sept 25-30 2005, pp. 89-92.

[2] Canfora, G. and Cerulo, L., "Impact Analysis by Mining Software and Change Request Repositories", in *Proceedings 11th IEEE International Symposium on Software Metrics (METRICS'05)*, September 19-22 2005, pp. 20-29.

[3] D'Ambros, M. and Lanza, M., "Software Bugs and Evolution: A Visual Approach to Uncover Their Relationships", in *Proceedings 10th European Conference on Software Maintenance and Reengineering*, 2006, pp. 227-236.

[4] D'Ambros, M., Lanza, M., and Gall, H., "Fractal Figures: Visualizing Development Effort for CVS Entities", in *Proceedings 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'05)*, Budapest, Hungary, 2005, pp. 46-51.

[5] Demeyer, S., Ducasse, S., and Nierstrasz, O., "Finding Refactorings via Change Metrics", in *Proceedings ACM Conference on Object-Oriented Programming, Systems, Languages and Application*, New York NY, 2000, pp. 166-178.

[6] Eick, S., Graves, T. L., Karr, A. F., Mockus, A., and Schuster, P., "Visualizing Software Changes", *IEEE Trans. on Software Engineering*, 28, 4, 2002, pp. 396 - 412.

[7] Eick, S., Steffen, J. L., and Summer, E. E., "Seesoft - A Tool For Visualizing Line Oriented Software Statistics", *IEEE Trans. on Soft. Engineering*, 18, 11, Nov. 1992, pp. 957-968.

[8] Froehlich, J. and Dourish, P., "Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams", in *Proceedings IEEE/ACM International Conference on Software Engineering (ICSE'04),* pp. 387-396.

[9] Gall, H., Jazayeri, M., Klosch, R., and Trausmuth, G., "Software Evolution Observation based on Product Release History", in *Proceedings IEEE International Conference on Software Maintenance (ICSM'97)*, pp. 160-166.

[10] Gall, H., Jazayeri, M., Krajewski, J., "CVS Release History Data for Detecting Logical Couplings", *Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, September 1 - 2, 2003 2003, pp. 13 - 23.

[11] Gall, H., Jazayeri, M., Riva, C., "Visualizing Software Release Histories: The Use of Color and Third Dimension", *IEEE International Conference on Software Maintenance (ICSM'99)*, August 30 - September 3, 1999 1999, pp. 99 - 108.

[12] German, D. M., "Mining CVS Repositories, the softChange Experience", in *Proceedings International Workshop on Mining Software Repositories (MSR'04)*, Edinburgh, Scotland, UK, 2004, pp. 17-21.

[13] Hassan, A., Mockus, A., Holt, R. C., and Johnson, P. M., "Special Issue on Mining Software Repositories", *IEEE Transactions on Software Engineering*, 31, 6, 2005, pp.

[14] Koike, H. and Chu, H. C., "VRCS: Integrating Version Control and Module Management using Interactive three-dimensional Graphics", in *Proceedings IEEE Symposium on Visual Languages (VL'97)*, Capri, Italy, 1997, pp. 170-175.

[15] Lanza, M. and Ducasse, S., "Understanding Software Evolution using a Combination of Software Visualization and Software Metrics", in *Proceedings Languages et Modeles a Objets (LMO'02)*, Lavoisier, Paris, 2002, pp. 135-149.

[16] Maletic, J. I. and Collard, M. L., "Supporting Source Code Difference Analysis", in *Proceedings International Conference on Software Maintenance*, September 11-17 2004, pp. 284-293.

[17] Marcus, A., Comorski, D., and Sergeyev, A., "Supporting the Evolution of a Software Visualization Tool through Usability Studies", in *Proceedings International Workshop on Program Comprehension*, St. Louis, MO, 2005, pp. 307-316.

[18] Marcus, A., Feng, L., and Maletic, J. I., "3D Representations for Software Visualization", in *Proceedings 1st ACM Symposium on Software Visualization (SoftVis'03)*, San Diego, CA, June 11-13 2003, pp. 27-36.

[19] Mattsson, M. and Bosch, J., "Observations on the Evolution of an Industrial OO Framework", in *Proceedings IEEE International Conference on Software Maintenance (ICSM'99)*, 30 Aug -3 Sept 1999, pp. 139-145.

[20] Storey, M. D., Čubranić, D., and German, D. M., "On the Use of Visualization to Support Awareness of Human Activities in Software Development: A Survey and a Framework", in *Proceedings 3rd ACM Symposium on Software Visualization*, St. Louis, Missouri, 2005, pp. 193-202.

[21] Tu, Q. and Godfrey, M. W., "An Integrated Approach for Studying Architectural Evolution", in *Proceedings 10th International Workshop on Program Comprehension (IWPC'02)*, Paris, France, 2002, pp. 127-136.

[22] Voinea, L., Telea, A., and Van Wijk, J. J., "CVSscan: Visualization of Code Evolution", in *Proceedings Symposium on Software Visualization,* St. Louis, MO, 2005, pp. 47-56.

[23] Wong, K., Blanchet, W., Liu, Y., Schofield, C., Stroulia, E., and Xing, Z., "JRefleX: towards supporting small student software teams", in *Proceedings OOPSLA Workshop on Eclipse Technology eXchange (EXT'05)*, 2003, pp. 50-54.

[24] Wu, J., Holt, R. C., and Hassan, A. E., "Exploring Software Evolution using spectrographs", in *Proceedings 11th Working Conference on Reverse Engineering*, 2004, pp. 80-89.

[25] Wu, X., Murray, A., Storey, M.-A., and Lintern, R., "A Reverse Engineering Approach to Support Software Maintenance: Version Control Knowledge Extraction", in *Proc. 11th Working Conf. on Reverse Engineering*, 2004, pp. 90-99.

[26] Zimmermann, T. and Weißgerber, P., "Preprocessing CVS Data for Fine-grained Analysis", in *Proceedings International Workshop on Mining Software Repositories (MSR 2004)*, Edinburgh, Scotland, U.K., 2004, pp. 2-6.

[27] Zimmermann, T., Zeller, A., Weißgerber, P., and Diehl, S., "Mining Version Histories to Guide Software Changes", *IEEE Trans. on Soft. Engineering*, 31, 6, Jun 2005, pp. 429-445.